Specification and verification of embedded systems
with Event B

J. Christian Attiogbé

Master ALMA

November 2017

UNIVERSITÉ DE NANTES

LS2N LABORATOIRE DES SCIENCES DU NUMÉRIQUE DE NANTES

Plan

Event-B: Some References

- *Modelling in Event-B: System and Software Engineering*, J-R. Abrial, Cambridge, 2010

- *Modelling and proof of a Tree-structured File System.* Damchoom, Kriangsak and Butler, Michael and Abrial, Jean-Raymond, 2008.

- *Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B.* Damchoom, Kriangsak and Butler, Michael; 2009.

- *Faultless Systems: Yes We Can!*, Jean-Raymond Abrial, 2009

- *Modelling an Aircraft Landing System in Event-B*, Dominique Méry, Neeraj Kumar Singh, 2014

- *Closed-Loop Modelling of Cardiac Pacemaker and Heart*, Dominique Méry, Neeraj Kumar Singh, 2012

Event B Specification Approach

Correct-by-construction: build correctly the systems (abstraction, modelling, refinement, composition/decomposition, proof)

Some hints to formal methods:

- Formal methods are rigorous engineering tools.
- Formal methods are means to build executable code from software requirement documents (informal, natural language).
- **Requirement Documents** (provided by clients) should be rewritten after analysis and understanding into Reference Document (where every thing is made clear and properly labelled for traceability).

B Method and Event B

- Event-B is an extension of the B-method (J-R. Abrial).
- It is devoted
  - for system engineering (both hardware and software), top-down approach
  - for specifying and reasoning about complex systems : concurrent and reactive systems.
- Event-B comes with a new modelling framework called **Rodin**. (like **Atelier B** tool for the classical B)
- The Rodin platform is an Eclipse-based open and extensible tool for B model specification and verification.
  It integrates various plugins: B Model editors, proof-obligation generator, provers, model-checkers, UML transformers, etc

Event B Modelling and its dissemination

Yet used in various case studies and industrial projects:

- Train signalling system
- Mechanical press system
- Access control system
- Air traffic information system
- Filestore system
- Distributed programs
- Sequential programs
- Cardiac Pacemaker
- etc

Event B Modelling: principles

Observe the behaviour of any system; what matters?

- A set of changes of its states.
- But, the observation distance does matter!
  (the details may be observed or not: parachutist paradigm)
- The observation focus does matter!
  (the observed changes are not the same)
- Different points of view = several abstractions.

Remind B Specification Approach

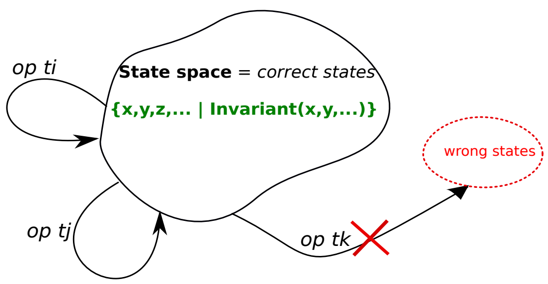


Figure: Do it right with B

```
VARIABLES
   x, y, z, ...
INARIANT
   Inv(x,y,z, ...)

OPERATIONS
   ti = ...
   tj = ...
   tk = ...
```

B Method: general development approach

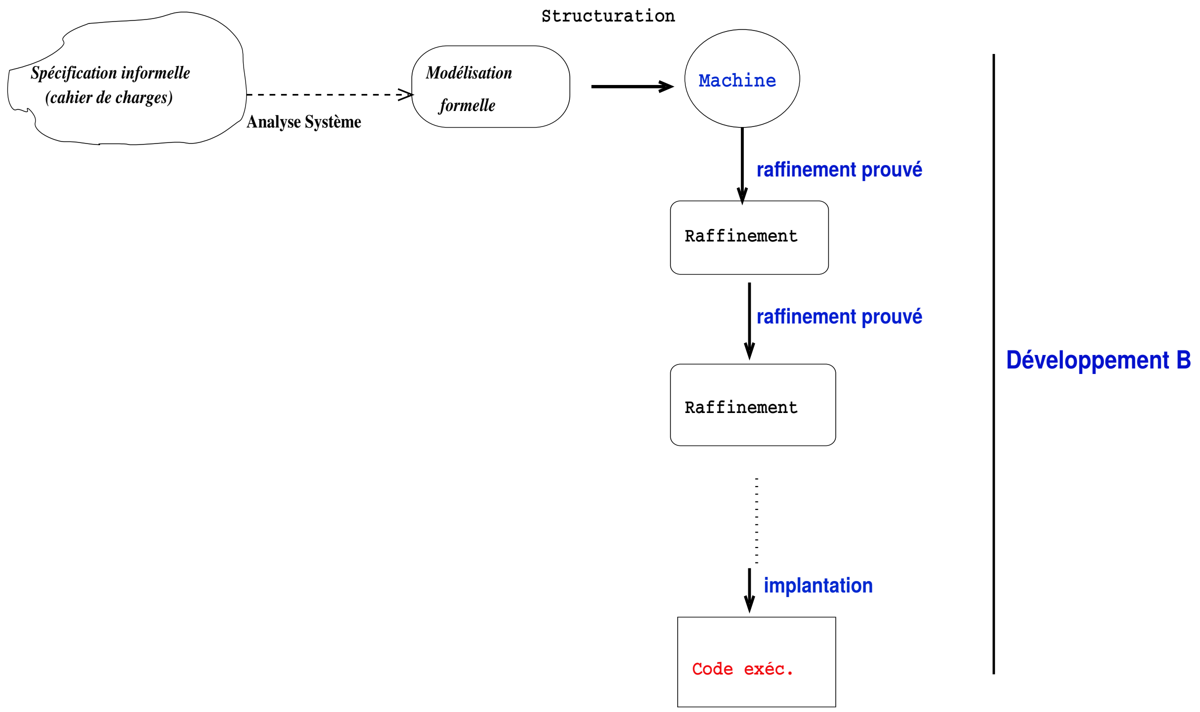


Figure: Development process with B

Event B Specification Approach

Event B Specification: start with Abstract system or Abstract model

An abstract system is a mathematical model of an asynchronous system behaviour

System behaviour: described by events which are observed!

Events are guarded actions/substitutions

Event occurrences involve a State-transition model.

A system model is a state-based model equipped with events

Event B Development Structuring

- Start with an Abstract system (or abstract model)
- Refinement of data and events
  The parachutist paradigm / microscope paradigm (JR Abrial)
- Decomposition (of a system into sub-systems, Hw, Sw)

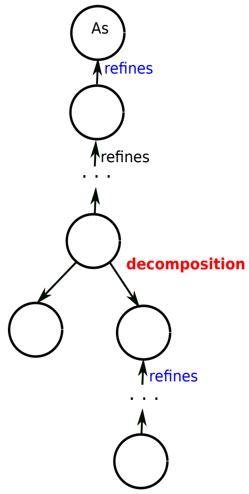


Figure: Event B Design structure

B Abstract System

Variables

Predicate

Events

```
SYSTEM
SETS ...
VARIABLES
    ...
INVARIANT
    ... predicate
INITIALISATION
    ...
EVENTS
    ...
END
```

but structured more efficiently using Contexts and machines.

Remind! Capturing the correct state space and events

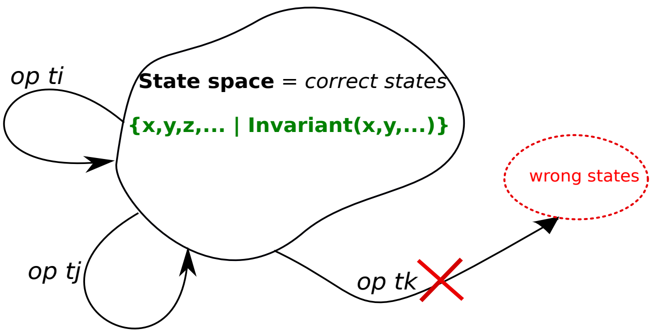


Figure: Events should preseve correct states

Capturing a system behaviour - Events

- The behaviour of a discrete system is a sequence of changes (system transitions).
- The changes may be internal or enabled by external signals.
- Each event describes the occurrence of a change in the discrete system under modelisation.
- Event B uses Guards and Actions [Dijkstra]

event = when Conditions then Effects

- But, the behaviour of a system may/should be captured gradually.

Formal Description of Events

An event has one of the following general forms (Fig. 5)

(WHEN/SELECT Form)	(ANY Form)
name $\widehat{=}$ /* event name */ WHEN /* the guard */ $P(gcv)$ THEN $GS(gcv)$ END	name $\widehat{=}$ /* event name */ ANY bv WHERE $P(bv, gcv)$ THEN $GS(bv, gcv)$ END

Figure: General forms of events

gcv denotes the global constants and variables of the abstract;
bv denotes the local bound variables of the event;
$P(bv, gcv)$ a predicate.

Formal Description of Events

An event without guard has the following form:

```
name ≙ /* event name */
  BEGIN
    GS(gcv)
  END
```

Abstract System (or a model, or a machine)

- The guard of an event with the WHEN form is: $P(gcv)$.
- The guard of an event with the ANY form is: $\exists(bv).P(bv, gcv)$.
- The WHEN form is a particular case of the other.
- The action associated to an event is modeled with a generalized substitution using the variables accessible to the event: $GS(bv, gcv)$.

Abstract System : Semantics and Consistency

An abstract system describes a mathematical model that simulates the behaviour of a system.
Its semantics arises from the invariant and is ensured by proof obligations (PO).
The consistency of the model is established by such proof obligations.

Consistency of an event B model

- PO: the initialisation establishes the invariant
- PO: each event of the abstract system preserves the invariant of the model

$I(gcv)$ the invariant and $GS(bv, gcv)$ the generalized substitution modelling the event action.

Abstract System : Semantics and Consistency

- the initialisation establishes the invariant;

$$[U]Inv$$

- each event preserves the invariant:
  In the case of an event with the ANY form, the proof obligation is:

$$I(gcv) \wedge P(bv, gcv) \wedge \text{prd}_v(S_e) \Rightarrow [GS(bv, gcv)]I(gcv)$$

Moreover the events (e) terminate:

$$I(Gcv) \wedge eGuard \Rightarrow \text{fis}(eBody)$$

(note $eBody \ = \ S_e$)

Introduction

Abstract System : Semantics and Consistency

The predicate fis(S) expresses that S does not establish $False$:

$$\text{fis}(S) \Leftrightarrow \neg\ [S]False$$

ie

$$I(Gcv) \wedge eGuard \Rightarrow \neg\ [S]False$$

The predicate prd$_v$(S) is the *before-after predicate* of the substitution S ; it relates the values of state variables just before (v) and just after (v') the substitution S, also written $BA_e(v, v')$.
The prd$_v$(ANY x WHERE $P(x, v)$ THEN $v := S(x, v)$ END) is:

$$\exists x.(P(x, v) \wedge v' = S(x, v))$$

Example: producer/consumer

Features: Concurrency and synchronization

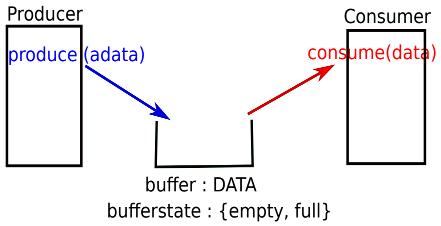


Figure: An overview of a producer-consumer

- Concurrent running of a process consumer which retrieves a data from a buffer filled by another process producer.
- The consumer cannot retrieve an empty buffer and the producer cannot fill in a buffer already full.

An event-driven model of the system is as follows:

Example : producer/consumer

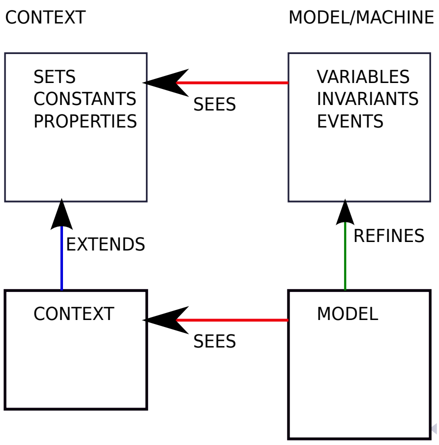
Machine *ProdCons* /* the abstract model */
sets
DATA ; STATE = {empty, full}
variables buffer, bufferstate, bufferc
invariants
bufferstate ∈ STATE ∧ buffer ∈ DATA ∧ bufferc ∈ DATA
initialization
bufferstate := empty || buffer :∈ DATA || bufferc :∈ DATA
events
produce $\widehat{=}$ /* if buffer empty */
any dd **where** dd ∈ DATA ∧ bufferstate = empty
then buffer := dd || bufferstate := full
end ;
consume $\widehat{=}$ /* if buffer is full */
select bufferstate = full
then bufferc := buffer || bufferstate := empty
end
end

Figure: A Producer-Consumer Abstract System

Structuring Event-B Models

An event-B model is structured with

- Contexts that contain carrier sets, axioms and theorems (seen by various machines)
- Machines which see the contexts and define a state space (static part: variables + labelled invariants) and a dynamic part made of some events.
- A context may be extended; a machine may be refined.



Refinement: principles

- Data refinement
  (as usually: new variables + properties; binding invariant)
- Event Refinement (extended):
  - Strengthening guards (unlike with Classical B)
    More variables are introduced with their properties.
  - Each event of the concrete system refines an event of the abstraction.
  - Introduction of new events which refine skip, and use new variables.

Refinement: principles

Let A with Invariant: I(av)

```
evt_a ≙ /* Abs. ev. */
   when P(av)
   then GS(av)
   end
```

avec $prd_v(...)$ = Ba(av, av')

Refined with: Invariant J(av,cv)

```
evt_r ≙ /* Conc. ev. */
   when Q(cv)
   then GS(cv)
   end
```

avec $prd_v(...)$ = Bc(cv,cv')

Proof obligation:

$$I(av) \wedge J(av, cv) \wedge Q(cv) \wedge Bc(cv, cv') \Rightarrow \exists cv'.(Ba(av, av') \wedge J(av', cv'))$$

Introduction

Event B Tools

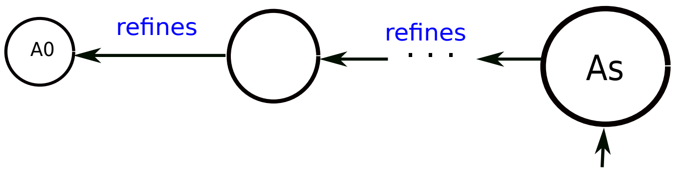
- First generation tools
  - Translation into classical B
  - B4free, Click'n'Prove
- New generation tools: DataBase, Eclipse Plugins, ...
  - Rodin (From sveral EU Projects: Matisse, Deploy, etc)

Refinement: structuring models

Refinement= development technique: various refinement strategies.

Horizontal refinement (feature augmentation)

From a small and abstract to a larger abstract model.
Details are gradually introduced in an abstract model in order to make it more precise
(wrt to requirements).



Refinement: structuring models

Vertical refinement: From abstrat to more concrete models

Details are gradually introduced in an abstract model
The specifier introduces new variables and makes some choices
Events may be split : event decomposition
machines may be split too: machine decomposition

Vertical Refinement: machine decomposition

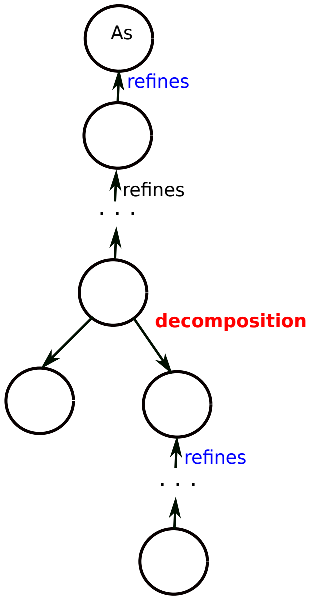
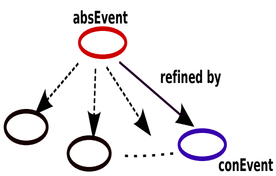


Figure: Vertical refinement with machine decomposition

Vertical Refinement: event decomposition

A coarse grain event is analysed and described in a more detailed (fine grain) way.
Think about the transfer of a file via a network.

- A given change consists of:
  start by sub-change...;
  follow by sub-change...;
  end by sub-change...;
- Hence, at least one sub-change (an event), refines the abstract event.



Machine Decomposition: structuring models

A coarse grain model is analysed and described in a more detailed (fine grain) way.
Think about a system involving software and physical devices.

- A given model is made of variables that model purely physical devices, and events are associated only to these variables
- The splitting is based on variables splitting (but not always straightforward).
- Divide and conquer: a small model is more tractable than a huge one.

Decomposition enables one to break complexity, to structure and develop more easily.

Machine Decomposition: structuring models

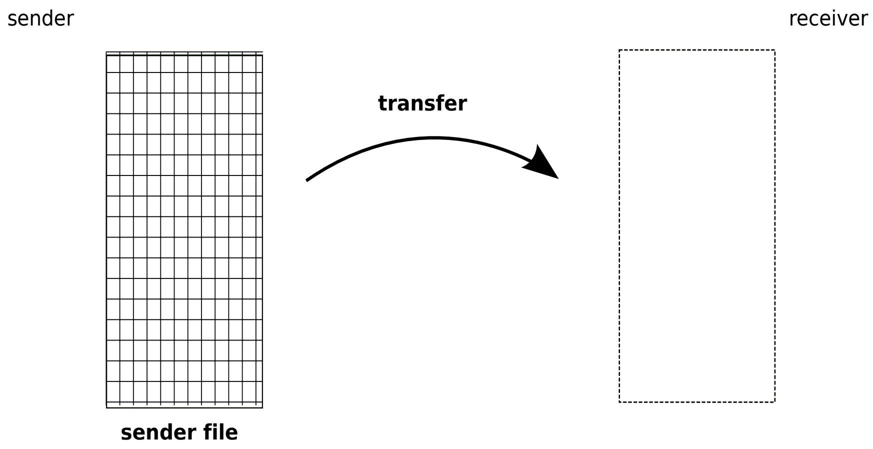
Machine variables and events are partitioned into sub-machines.

- Decomposition with Abrial's style (shared variables): the sub-machines may interact with each other via shared variables. Shared variables are duplicated, new external-events are introduced in each machine that has a shared variable in order to ensure consistency of changes.
- Decomposition with Butler's style: the variables are not shared; an event which uses variables in separate machines, is shared (then separated-duplicated).
  The sub-machines may interact with each other via synchronisation over shared parameterised events.

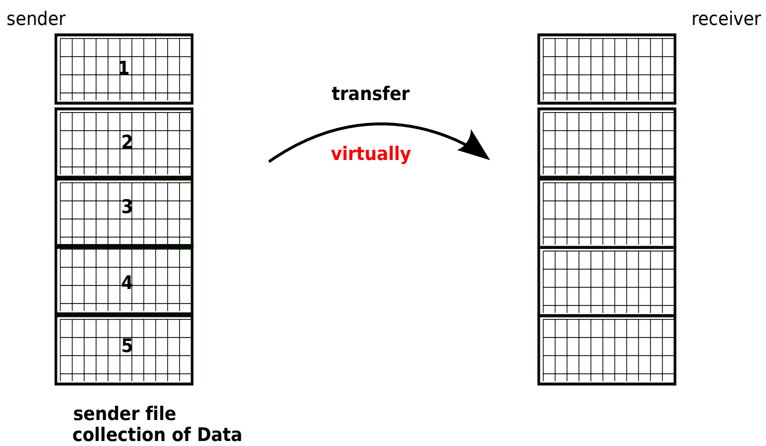
Event-B Model Decomposition, C. Pascal(Systerel), R. Silva(Univ. of Southampton)

Event-B Model - Example: File transfer protocol

Specification of a file transfer between two sites: a sender and a receiver.

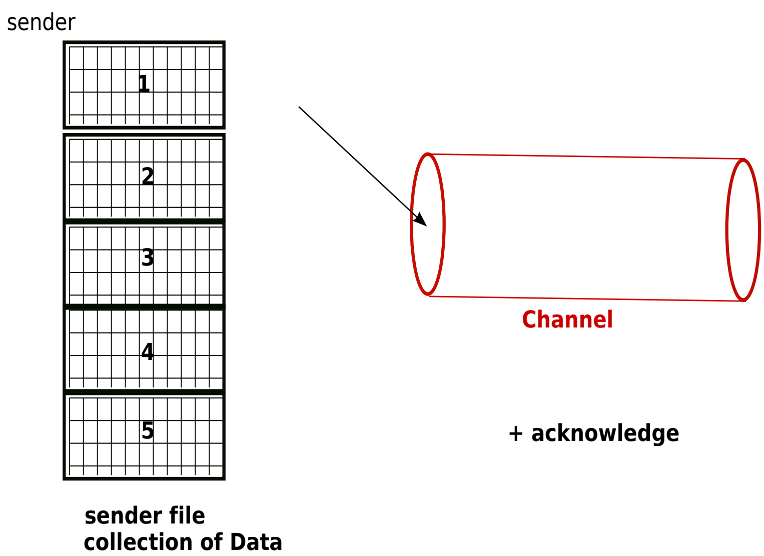


Event-B Model - Example: File transfer protocol



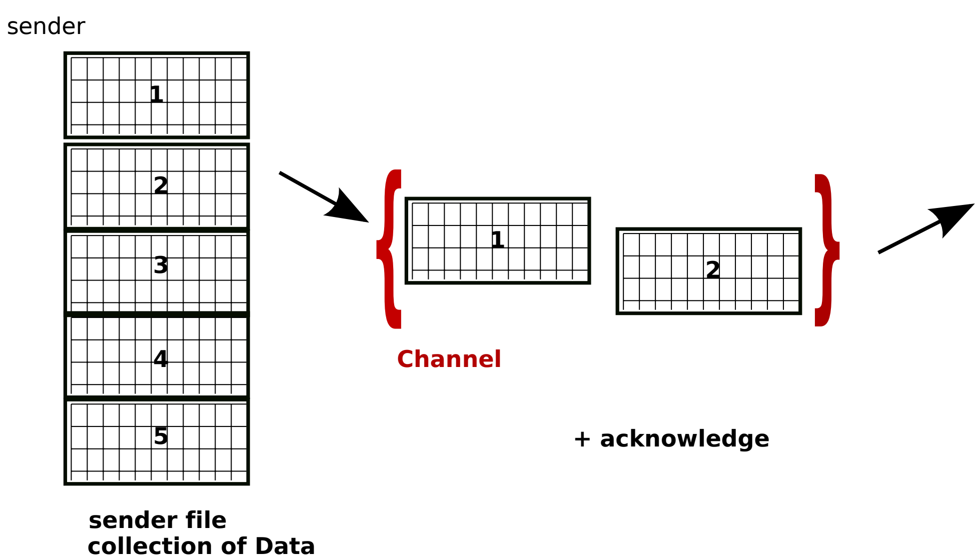
A file is made of a set of data records.
From a very abstract level, the transfer is done instantaneously.

Event-B Model - Example: File transfer protocol



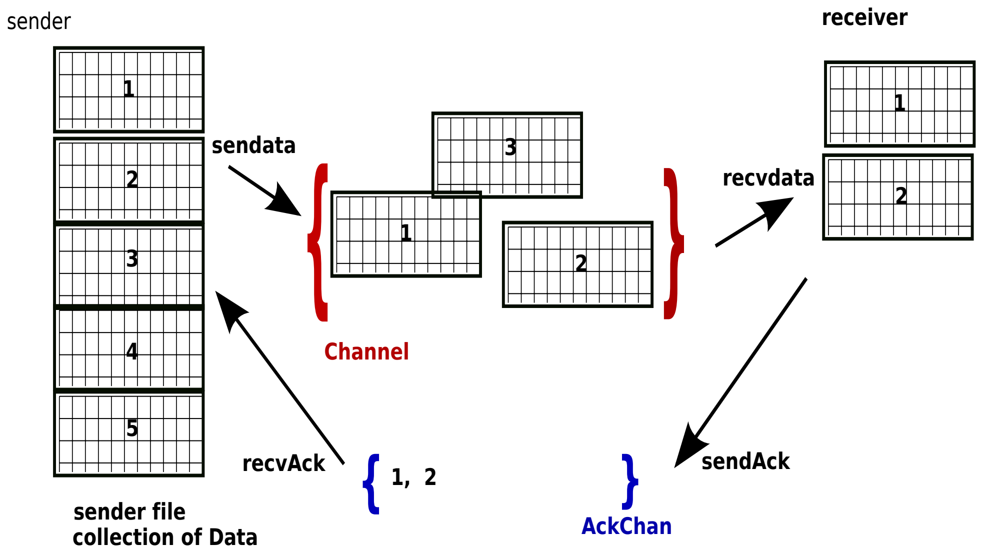
But, a file is made of a set of data records which are to be transfered through a channel.

Event-B Model - Example: File transfer protocol



From a more concrete level, the transfer is achieved step by step, one record after the other.

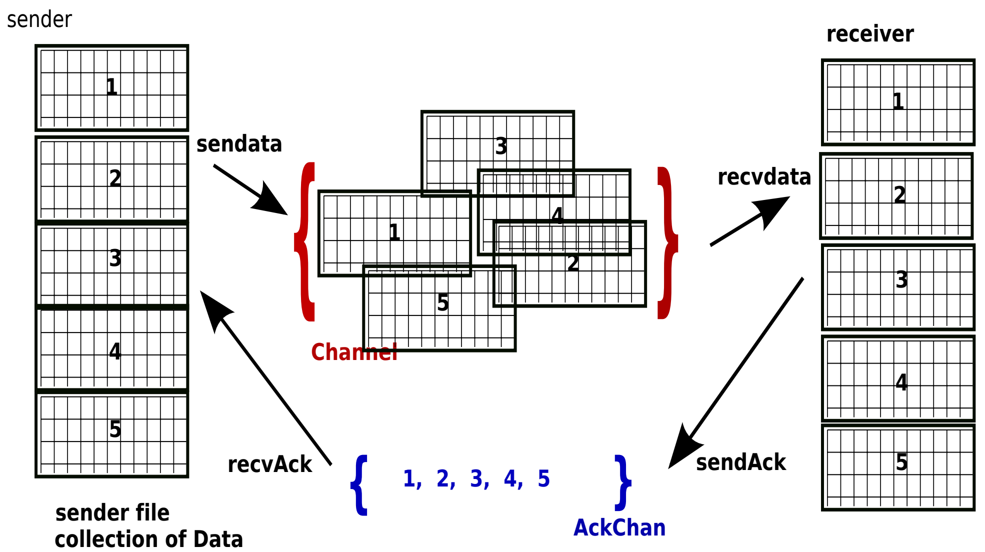
Event-B Model - Example: File transfer protocol



There are some intermediary operations, to send data on the channel from the sender side, to receive data from the channel from the receiver side. In the same way acknowledgements are sent/received.

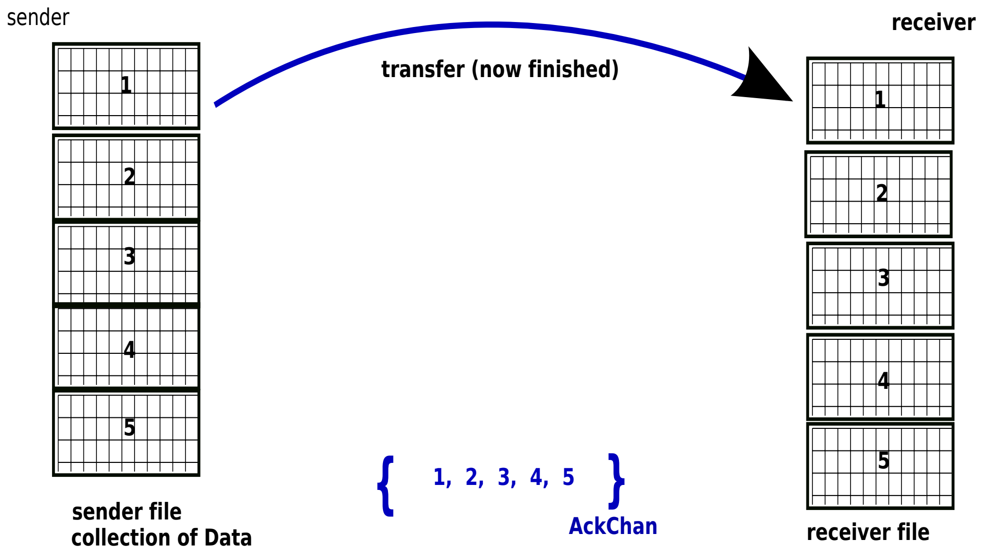
Examples - Case studies

Event-B Model - Example: File transfer protocol



Only after all the intermediary operations, the transfer will be completed.

Event-B Model - Example: File transfer protocol



Examples - Case studies

Event-B Model - Example: File transfer protocol

- Senderfile = some data records = $1..nr \rightarrow DATA$
  $\{1 \mapsto data1,\ 2 \mapsto data2\ , \cdots\}$
- A channel is a set of such data records.
- At each time, the channel contains a part (set inclusion) of the sender's file
- The receiver acknowledges the received records numbers.
- The file transfer is completed when all the records are acknowledged.
- Failure: loss of data/ack in the channels.

We have the model!

Event-B Model Example: File transfer protocol

```
MACHINE Transfer
SETS DATA
CONSTANTS nr /* file size : number of records
*/
PROPERTIES nr : NAT & nr > 1
VARIABLES
sf /* sender file */
, rf /* receiver file */

INVARIANT
& sf : 1..nr -> DATA /* all records of sf */
& rf : 1..nr +-> DATA /* probably part of
records of sf */
INITIALISATION
sf := {} || rf := {}
```

```
EVENTS
transf = /* instantaneous transfer, from far
way */
BEGIN
rf := sf
END

/* but, technically, we will need to anticipate
the intermediary events */
END
```


Event-B Model Example: File transfer protocol

```
MACHINE Transfer
SETS DATA
CONSTANTS nr /* file size */
PROPERTIES nr : NAT & nr > 1
VARIABLES
sf /* sender file */
, rf /* receiver file */
INVARIANT
& sf : 1..nr -> DATA /* all records of sf */
& rf : 1..nr +-> DATA /* probably part of
records of sf */
INITIALISATION
sf := {} || rf := {}
```

```
EVENTS
transf = /* instantaneous transfer, from far
way */
BEGIN
rf := sf
END

/* the following events are introduced by
anticipation of the forthcoming gradual
refinement*/
; sendta = skip
; recdta = skip
; sendac = skip
; recvac = skip
/* the followings are events that simulate the
non-releiabiliy of channels */
; rmvData = skip
; rmvAck = skip
END
```

Event-B Model Example: File transfer protocol

```
REFINEMENT
Transfer_R1

REFINES Transfer

VARIABLES

cs /* current record to be sent */
, cr /* current record received */
, rf
, sf /* sender file */
, erf /* effectively received file */
, dataChan /* data channel */
, ackChan /* ack channel */
 INVARIANT
cs : 1..nr+1 /* current to be sent */
& cr : 0..nr /* current received */
& cr <= cs /* current received is <= current
sent */
& cs <= cr+1 /* cr <= cs <= cr+1 */
& erf = (1..cr) <| sf
& dataChan <: (1..cs) <| sf
& ackChan <: 1..cr
```

```
INITIALISATION
cs := 1
|| cr := 0
|| rf := {}
|| sf := {}
|| erf := {}
|| dataChan := {}
|| ackChan := {}
EVENTS
transf =
WHEN
cs = (nr + 1) /* that is all cs are received
(last ack received) */
THEN
rf := erf /* not necessary, effective copy of
the received file in the receiver */
END

... (continued)
END
```


Event-B Model Example: File transfer protocol

```
/* new events introduced (ie. we "forget" the
anticipation in the abstract model) */
; sendta =
WHEN
cs <= nr
THEN
dataChan(cs) := sf(cs)
/* now wait for the ack, before updating cs */
END

; recdta =
WHEN cr+1 : dom(dataChan)
THEN
erf(cr+1) := dataChan(cr+1)
|| cr := cr + 1 /* the next data to be received
*/
END

; sendac =
WHEN cr /= 0 /* send ack for the received cr
data */
/* may be observed repeatedly until the next
data */
THEN
ackChan := ackChan  {cr}
END
```

```
recvac =
WHEN cs : ackChan /* ack for the already sent
cs */
THEN
cs := cs + 1 /* now the next to be sent */
END
/* Simulating non-relaibility of channels,
data/ack may be loss */
; rmvData =
ANY ii, dd WHERE
ii |->dd : dataChan
THEN
dataChan := dataChan - { ii|->dd }
END
;
rmvAck =
ANY ii WHERE
ii : ackChan
THEN
ackChan := ackChan - {ii}
END
```

Case Study : Multiprocess specification (Readers/writers)

- Description
  - Multiple processes: readers, writers
  - Shared resources between the processes
  - Several readers may read the resource
  - Only one writer at a time
- Property:
  Mutual exclusion between readers and writers
- Improvement:
  no starvation → as a new property
  (using refinements)

Multiprocess specification

```
MACHINE
readWrite2
SETS
WRITER /* set of writer processes */
; READER /* set of reader processes */

VARIABLES
writers /* current writers */
, activeWriter
, waitingWriters
, readers /* current readers */
, waitingReaders
, activeReaders /* we may have svrl readers simultan. */
```

Multiprocess specification

```
INVARIANT
writers <: WRITER
& activeWriter <: WRITER & card(activeWriter) <= 1
& waitingWriters <: WRITER
& writers /\ waitingWriters = {}
& activeWriter /\ waitingWriters = {}
& activeWriter /\ writers = {}
/* merge */
& readers <: READER
& waitingReaders <: READER
& activeReaders <: READER    &    card(activeReaders) >= 0
& readers /\ waitingReaders = {}
& activeReaders /\ waitingReaders = {}
& activeReaders /\ readers = {}
/*-----safety properties -----*/
& not((card(activeWriter) = 1)&(card(activeReaders) >= 1))
```


Case study: readers-writers

Multiprocess specification

```
INITIALISATION
activeWriter := {}
|| waitingWriters := {}
|| activeReaders := {}

|| readers :: POW(READER)
|| writers :: POW(WRITER)
|| waitingReaders := {}
```

Multiprocess specification

```
want2write = /* observed when a process wants to write */
ANY ww WHERE
ww : writers
& ww /: waitingWriters
& ww /: activeWriter
THEN
waitingWriters := waitingWriters \/ {ww}
|| writers := writers - {ww}
END
;
writing =
ANY ww WHERE
ww : waitingWriters
& activeReaders = {} & activeWriter = {}
THEN
activeWriter := {ww}
|| waitingWriters := waitingWriters - {ww}
END
```


Case study: readers-writers

Multiprocess specification

```
endWriting =
ANY ww WHERE
ww : activeWriter
THEN
writers := writers\/ {ww}
|| activeWriter := {}
END
;
want2read =
ANY rr WHERE
rr : readers
& rr /: waitingReaders
& rr /: activeReaders
THEN
waitingReaders := waitingReaders \/ {rr}
|| readers := readers - {rr}
END
```

Multiprocess specification

```
reading =
ANY rr WHERE
rr : waitingReaders
& activeWriter = {}
THEN
activeReaders := activeReaders\/ {rr}
|| waitingReaders := waitingReaders - {rr}
END
;
endReading =
/* one of the active readers finishes and leaves
the competition to the shared resources */
ANY rr WHERE
rr : activeReaders
THEN
activeReaders := activeReaders - {rr}
|| readers := readers \/ {rr}
END
```


Multiprocess specification

```
newWriter = /* a new Writer */
ANY ww
WHERE ww : WRITER
& ww /: (writers \/ waitingWriters \/ activeWriter)
THEN
writers := writers \/ {ww}
END
; leaveWriters = /* a writer leaves the group */
ANY ww
WHERE
ww : writers
THEN
writers := writers - {ww}
END
```

Multiprocess specification

```
newReader = /* a new reader joins the readers */
ANY rr WHERE
rr : READER
& rr /: (readers\/waitingReaders \/activeReaders)
THEN
readers := readers \/ {rr}
END
; leaveReader =
ANY rr WHERE
rr : readers & card(readers) > 1
THEN
readers := readers - {rr}
END
```


Conclusion

- Initiation rapide à B et Event-B
- Reste à pratiquer, pratiquer, et pratiquer encore

Event-B: Some References

- *Modelling in Event-B: System and Software Engineering*,
J-R. Abrial, Cambridge, 2010

- *Modelling and proof of a Tree-structured File System.* Damchoom, Kriangsak and Butler, Michael and Abrial, Jean-Raymond, 2008.

- *Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B.* Damchoom, Kriangsak and Butler, Michael; 2009.

- *Faultless Systems: Yes We Can!*, Jean-Raymond Abrial, 2009

- *Modelling an Aircraft Landing System in Event-B*,
Dominique Méry, Neeraj Kumar Singh, 2014

- *Closed-Loop Modelling of Cardiac Pacemaker and Heart*, Dominique Méry, Neeraj Kumar Singh, 2012